Haskellによる複数移動体の協調動作制御に関する研究

Haskellによる複数移動体の協調動作制御に関する研究
Study of coordinated operation and control for multiple object by Haskell

堀　麻衣
Asagi Hori
法政大学情報科学研究科情報科学専攻
E-mail: asagi.hori.2r@stu.hosei.ac.jp

Abstract

This paper describes about study of coordinated operation and control for multiple object by Haskell. Haskell, which is a functional programming language, is said to be better than imperative language in logical description and calculation. Haskell does not have many practical examples to real systems, but Haskell has possibility to improve abstraction level in programs and to provide accurate and reliable programs. In this study, line trace car which is added stop function and following robot for line trace car are implemented by Haskell, and designed robot simulators by Haskell and Yampa which is functional reactive programming, and are verified the possibility of applying to a real system. In this experiment, robot programs and robot simulators outputted motor's speeds correctly in response to inputs from sensors. However, trouble that real sensors and motors is not set upped are also observed. This trouble will can be solved by correcting interfaces to connect Haskell and real systems. It is also possible to improve programs as simply. Though coordinated operation and control for multiple object has not been fully realized in this study, further study is continued to apply as simply by correcting interfaces and updating software and hardware.

Key words: Coordinated operation and control, Multiple object, Haskell, Functional Reactive Programing (FRP), Yampa,

1. まえがき

本論文では，Haskell による複数移動体の協調動作，および制御の研究について述べる．純関数型言語の Haskell は，論理的な記述および計算，また並列処理を行うことに優れた機能を有している．しかし外部からの入力および外部への出力，実システムへの応用例に乏しく，またそのために必要なライブラリの種類も不十分である．それゆえ，特にロボットのシステムに応用される例は未だに少ない．本研究では，Haskell の関連言語および関連技術を用いることにより，実システムへの応用の可能性を示す．具体的には，Haskell によって異なる機能を実装した複数の移動体の実装を行い，シミュレーションプログラムによって実システムへの応用の可能性を検証する．

本研究で用いられる Haskell は，逐次処理的な記述法（how 方式）をとる C 言語や java などの手続き型言語と異なり，数学的および圏論に則った構造的な記述法（what 形式）を導入している．この記述法はプログラムの抽象度を向上させることが可能であり，参照透過性や比較的シンプルな記述法による可読性の高さからロボット制御において正確性と信頼性の高いプログラムの実装を行える可能性がある．

Haskell の大きな特徴としては，圏論の概念を利用できることが挙げられる．圏論の概念を利用することにより，本来は参照透過性を持つ言語では実装できない値の代入などの副作用を持つ操作を参照透過性を壊すことなく実装できる．これにより，Haskell は参照透過性を持つ純関数型言語でありながら，手続き型言語のような記述でのプログラミングができる[1].

以上のことから，ロボット操作および制御において Haskell は参照透過性および可読性の向上の面から信頼性の高いプログラムの実装を行う目的で利用でき，実システムにおいてより複雑な計算を速く正確に行い，複雑なながらも安定した動作が実現できると考えた．

Haskell のプログラムは参照透過性により，関数は最初に与えられた値しか使用できないという特性を持っている．また，Haskell だけではセンサーや入力デバイスによる外部からの入力，グラフィックやモーター出力などの外部への出力を行うことができない．そこで，本研究では他言語による外部インタフェースの併用および関数型リアクティブプログラミングによる外部との入出力を試みた．他言語による外部インタフェースを利用することにより，Haskell のプログラミングとハードを繋ぐ．関数型リアクティブプログラミングの利用は，外部からの時間と共に変化する入力を直接保存し，関数の入出力の値として渡すことが可能となる．

今回は Brick Pi を用いた検証には C 言語による外部インタフェースを利用し，ハードとの接続を行わないシミュレーションプログラムには Haskell の派生言語である Yampa[2]を関数型リアクティブプログラミング(FRP)として利用した．

Supervisor: Prof. Nobuhiko Koike

2. 複数移動体のシステム構成

本研究で想定する複数移動体は異なる条件によりモーターの速度調節を行う二体のロボットである．一体は地面を走行する際に左右のカラーセンサーによって地面の色を判別し，モーターの速度を調節するライントレースカーである．もう一体は一体目のライントレースカーをライントレースの機能なしで追従するロボットである．

ライントレースカーおよび追従ロボットのシステム構成は以下の通り．

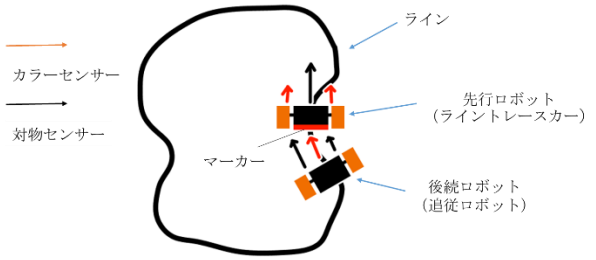

図１　複数移動体のシステム構成

先行するロボットの後部にはあらかじめマーカーをつけ，後続する追従ロボットはマーカーの色で先行するロボットを認識し，先行するライントレースカーとの距離に応じて速度を変更させながら追従を行う．

2. 1. ライントレースカー

ライントレースカーはロボットの左右にカラーセンサーを搭載し，床面の色のデータを入力値として渡す．センサーから与えられた入力値により床面の色が白と判断された場合はモーターの回転数を 100/秒に，黒と判断された場合は回転数を 50/秒にする．これにより，モーターの速度を変更させて軌道修正を行いながら，白い紙の上に描かれた黒線のレールに沿うように直進や曲がる動作をすることが可能となる．今回使用するライントレースカーは左右の床面が黒と判断した場合は左右両方のモーターの回転数を 50/秒にして直進する動きを実装しているが，本来は Don’t care の状態として処理するものである．これに加え，ロボットの正面部に対物センサーを搭載することにより，ロボットの前に障害物がある場合は停止の動作を行う．

ロボット内で行われる入出力の処理の流れは以下の通り．

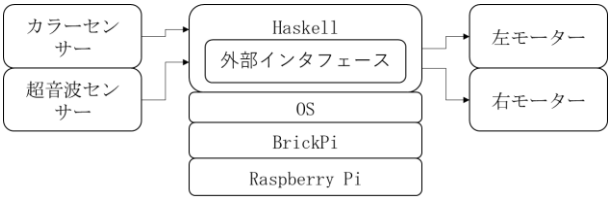

図２　ロボット内の入出力の流れ

2. 2. 追従ロボット

もう一体は，一体目のライントレースカーを追従するためのロボットである．このロボットはライントレースカーと同様に左右のモーターの速度を調節して動くが，モーターへの出力までの処理の流れはライントレースカーと異なる手法をとる．追従を行うロボットはあらかじめ先行するライントレースカーに付けられたマーカー（本研究では赤いマークを目印とする）をカラーセンサーで識別し，追従する．ロボットの正面部にカラーセンサーを，左右に超音波センサーを搭載し，前方に印の赤色があると判断した場合は対物センサーが判別した前方の物体の距離によってモーターの速度を調節する．距離が近すぎる場合はモーターを停止させ，特定の距離の範囲内にいる場合はモーターの回転数を 100/秒にし，特定の距離の範囲から離れた場合はモーターの回転数を 200/秒にする．また，追従するロボットの印である赤色が判別できない場合は前方の物体の種類に関わらず左右のモーターを停止させる．

本研究では，ライントレースカーの停止命令および追従ロボットのモーター制御のために超音波センサーを搭載している．超音波センサーは計測できる範囲にある物体との距離を入力値として渡し，計測する対物センサーとしての役割を果たす．ライントレースカーの速度調節は対物センサーによって一時停止の処理が起こらず，かつ左右の色情報がある場合にのみ起こるものとする．また，追従ロボットにおいては先行するロボットに遅れをとって見失わず，かつ追従する際に先行するロボットとの距離が短すぎることによる衝突を防ぐ目的で使用される．

3. Haskell による実システムへの実装について

表１に本研究で使用した資源の一覧を示す．

表１　研究に使用した資源一覧

Model	Version
Raspberry Pi	Raspberry Pi 2 model B Raspberry pi 3
BrickPi	BrickPi BrickPi 3
LEGO Mindstorms Sensors	LEGO Mindstorms EV3
LEGO Mindstorms Motors	LEGO Mindstorms NXT
OS	2015.03.20_Dexter_Industries_wheezy(Raspbian) 2017.10.05- BETA_Dexter_Industries_jessie(Raspbian)
GHC	7.6.3
Yampa	0.10.7
GLUT	3.7

本研究では Haskell によるライントレースカーおよび追従ロボットの実装を行うため，実際に動くセンサーとモーターを搭載したハード，および操作と制御のためのコンピュータとこれらを繋ぐためのキットを用意する必要がある．本研究では，操作および制御のためのコンピュータとして Raspberry Pi を使用し，モーターおよびセンサーは Lego Mindstorms のものを使用する．ただし，Lego Mindstorms は Raspberry Pi および Haskell に直接対応して

いるハードではないため，これらを繋ぐキットとして Brick Pi を搭載した．

3.1. Raspberry Pi の準備

初期状態の Raspberry Pi では Haskell のプラットフォームがないため，あらかじめ以下の手順によって Haskell を使用できる環境を揃える．なお，一部の手順および遠隔操作を行うために Tera Term および win-SCP を使用し，PC からインターネット回線を経由して Raspberry Pi の操作およびファイルの転送を行う．[7]

1. 用意した OS のイメージファイルを実装用の SD カードに書き込む．この際，イメージファイルは DD for Windows を用いて書き込みを行った．書き込みが終わった後，コマンド画面で Raspberry Pi のコンフィグレーションを開き，ファイルシステムの拡張を行う．
2. コンフィグレーション画面で SSH を有効化する．この操作により Tera Term と win-SCP による操作が可能となる
3. SD カードの空き領域をハードディスクとして利用できるよう設定し，以下のコマンドから OS のアップデートとアップグレードを行う．
   sudo apt-get update
   sudo apt-get upgrade
4. 以下のコマンドによって Haskell のプラットフォームをインストールする．
   sudo apt-get install ghc Haskell-platform
   Haskell のプログラムによってモーターやセンサーを使用する際，Concuttent-extra のライブラリが必要になるため，以下のコマンドによってインストールを行う．このライブラリは，スレッドの遅延効果を使用するためのものである．
   cabal update
   cabal install concurrent-extra

3.2. BrickPi およびプログラムの準備

Raspberry Pi に Haskell のプラットフォームを揃え，BrickPi と Raspberry Pi をピンで接続し，各センサーとモーターを LAN ケーブルによって BrickPi に接続することで，図３のようなロボットとなる．

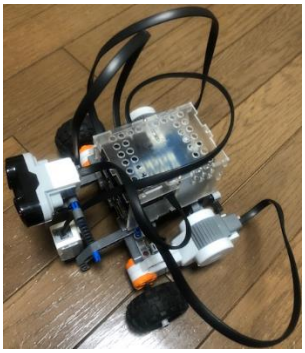

図３　ロボットの全体

また，ロボットにおける各パーツの関係は図４の通りである．

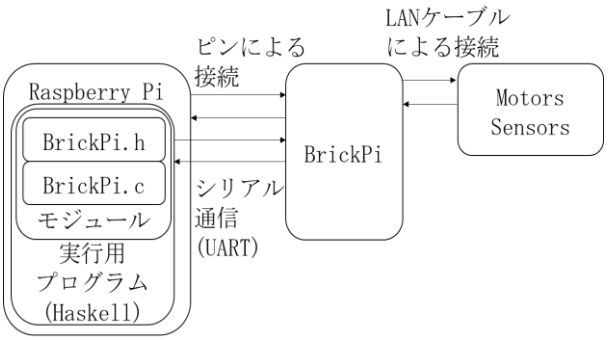

図４　ロボットにおける各パーツの関係

ロボットの操作と制御を行うプログラムは Haskell によって書かれるが，BrickPi は Haskell で書かれた命令には対応できないため，実機でプログラムを実装するために C 言語で書かれた外部インタフェース[3]を使用する．使用した外部インタフェースの概要は以下の通り．

・BrickPi.c, BrickPi.h　プログラムを実行する際に，モーターおよびセンサーの制御データを設定するために使われる外部インタフェース

・tick.c, tick.o　プログラムを実行する際の時間制御に使われる外部インタフェース

また，外部インタフェースを定義するためのモジュールを別途用意する．

Haskell で書かれた操作および制御プログラムは，以下のコマンドによってコンパイルし実行用プログラムを作成する．

ghc --make プログラム名.hs BrickPi.o tick.o

上記のコマンドの BrickPi.o および tick.o は BrickPi.c および tick.c に変更が可能である．また，実行用プログラムを起動する際は以下のコマンドより実行する．

./プログラム名

実システムおよびシミュレーターのプログラムに共通して実装する主な関数は以下の通り．

・initialize　ロボットの初期化に関わる関数．BrickPi 本体をセットアップし，各ポートに接続されたセンサーおよびモーターのセットアップを行う．

・lineSensors および ultrasonicSensors　各センサーから入力値を受け取り，入力値によってモーターの速度調節あるいは一時停止の判定に必要な情報を渡す．ライントレースカーの場合は lineSensors によって速度の調節を行い，ultrasonicSensors によって一時停止の判定を行う．一方追従ロボットの場合は lineSensors によって一時停止の判定を行い，ultrasonicSensors によって速度の調節を行う．

・motors　一時停止と速度調節の判定に必要な値を引数として受け取り，左右のモーターの速度の値を同時に設定する関数．実システムに組み込むプログラムの場合，速度の設定をした後に値の更新を行う記述を加える．

以上の内容を実装したプログラムをコンパイルして実行用プログラムを作成し，その後プログラムを実行して検証を行う．

4. Haskell によるシミュレーターの実装

本研究では実システムによる検証を試みると同時に，実システムで実行できない場合に操作および制御用プログラムがどのような挙動をするか検証するシミュレーターの実装を行った．シミュレーターはセンサーの値を直接受け取らず，入力値となるセンサーの値を直接入力し，それに応じたモーターの挙動を文字および視覚情報で示すものとする．このシミュレーターはセンサーから入力値を受け取ることに加え，ロボットを監視する人間およびPCがロボットに渡すべき情報を直接伝えた場合を想定する．

4. 1. Haskell を用いたシミュレーションの実装

Haskell のみを用いたシミュレーションでは，センサーからの入力の代わりに直接入力される情報を入力値として渡し，入力値に対応したモーターの速度を表す値をコンソールに表示する．実システムで使用するプログラムには Haskell のみでは使えない記述や関数があるので適宜シミュレーション用に変更を行うが，シミュレーションを行う際は実システムで実装したプログラムと同じ関数が使用される．

実システムでは main 関数で initialize 関数を呼び出し，センサーやモーターの入出力を処理する関数へいこうしたが，シミュレーションではテストを行う関数の流れでセットアップと入出力用の関数を呼び出し，シミュレーションを行う．

なお，Haskell においてコンソールからの入力を行う際は以下の記述によってモジュールを入れる必要がある．

```
import Control.Applicative
```

4. 2. 関数型リアクティブプログラミング(FRP)を用いた実装

Haskell のみを用いたシミュレーションプログラムを実行した場合，出力の値は文字のみで表示される．そこで，FRP を用いたプログラムを併せて実装することにより，シミュレーション結果を図として描画することにより視覚的に表示する．なお，Yampa だけ用意しても描画は行えないため，別途 GLUT[4]をインストールしてインポートする必要がある．

Yampa を始めとした関数型リアクティブプログラミングの特徴は，プログラミング中に時間の概念を取り入れ，時間 t という連続的に値が変わる変数を用いた記述が関数型言語でも可能になることである．なおこの時間 t は命令実行時からの時間の値を指す．これにより，連続的な時間を用いた積分の数式を取り入れることが可能となった．

また，Yampa は信号関数とアロー記法を使用する記述も特徴として挙げられる．信号関数は以下のように記述される．[5][6]

$$SF\ A\ B \approx (t \to A) \to (t \to B)$$

この関数は時間 t によって連続的に変化する信号 A を時間 t によって連続的に変化する信号 B に写像する．F は Δt を要約したものとなり，信号 B が信号 A と F に依存する場合は純粋でない関数となり，信号 B が信号 A にしか依存しない場合は純粋な関数となる．

アロー記法はモナドと呼ばれる数学的な概念に関する抽象データ型のインタフェースであり，計算を表現する型に対して適している．アロー記法は基本的に”出力値<-関数<-入力”という形式の記述を行っており，処理は不可逆である．

今回用いたアロー記法の主なオペレータは以下の通り．[5]

・arr :: 持上げ(Lifting) 普通関数をアロー記法のオブジェクトに変換する

・(>>>) :: 合成(Composition) 一連の関数を合成する

なお，アロー記法を用いる場合は文頭の記述を以下のものにする必要がある．

```
{-# LANGUAGE Arrows #-}
```

本研究で実装したプログラムでは，センサーからの入力からモーターの速度を出力までのメソッドは Haskell のみで記述したプログラムをベースとし，モーターの出力を図として視覚的に表現する命令を Yampa と Haskell で実装する．

このプログラムでは，描画のために以下の型シノニムを設定する

・type Pos = (Double, Double) 描画のための x 座標と y 座標を設定するための型シノニム．GLUT の描画は三次元の座標が必要になるが，このシミュレーターは二次元的な描画を行う仕様であるため，z 座標は定数のまま x 座標と y 座標を型シノニムとして設定した．

・type Vel = (Double, Double) モーターの速度を設定し，描画するための型シノニム．出力されたモーターの速度を描画するオブジェクトに反映させ，座標を求める式に代入する．

・type R = GLdouble 描画するオブジェクトに色を付けるための型シノニム．入出力に関わらない部分にのみ使用する．

上記の型シノニムとモーターへの出力値を併せて描画を行うために用いた関数は以下の通りである．

```
output :: Pos -> Vel -> SF () (Pos, Vel)
output pos0 vel0 = proc input -> do
  vel <- (vel0 ^+^) ^<< integral -< (ax, ay)
  pos <- (pos0 ^+^) ^<< integral -< vel
  returnA -< (pos, vel)
```

この関数は座標の情報と速度の情報を入力値とし，時間 t による積分を計算し，モーターの速度と座標を求めた結果を出力値として返す．関数内で行われている計算は以下の通り．

$$Vel(t) = vel0 + \int_0^t a\ dt$$

$$Pos(t) = pos0 + \int_0^t vt\ dt$$

関数 output は上記の速度に対する積分と座標に対する積分を x 座標と y 座標に関して一括で計算している．上記の関数において ax と ay を x 座標と y 座標における加速度 a として定数を置いているが，今回のシミュレーターでは左右のモーターの速さに応じてオブジェクトが等速度直線運動を行う仕様のため，ax と ay には 0 を代入している．

```
doingOutput :: Pos -> Vel -> SF () (Pos, Vel)
doingOutput pos0 vel0 = switch (bb pos0 vel0) (\ (pos, vel) -> if snd pos < 0 then doingOutput ((fst pos), 10) vel else doingOutput pos vel
  where bb pos0' vel0' = proc input -> do
        (pos, vel) <- output pos0' vel0' -< input
        event <- edge -< snd pos <= 0
        returnA -< ((pos, vel), event `tag` (pos, vel))
```

この関数は入力値と時間 t を更新しつつ，更新した入力値と時間 t を用いて output 関数を呼び出す．output 関数を呼び出す前に if 文によって呼び出す際に渡す値の内容を調整している．今回は表示する情報の仕様上，y 座標が 0 を下回るときに y 座標が初期値に戻るよう条件付けをし，そうでない場合は入力値をそのままに output 関数を呼び出す．

```
simulate :: SF (Event ()) (IO ())
simulate = discardInputs >>> update >>> draw
```

main 関数から呼び出される，描画までのオブジェクトの状態の設定から描画までを行うための関数．discardInputs 関数から update 関数の間にオブジェクトの状態の設定，速度と座標の初期設定から更新を行い，draw 関数で結果を描画する．なお，今回の実装では手続き型言語におけるグローバル変数を設定できなかったので，モーターの速度に応じた simulate 関数，draw 関数，main 関数を用意している．

```
idle :: IORef Int -> ReactHandle (Event ()) (IO ()) -> IO ()
idle oldTime rh = do
  newTime'  <- get elapsedTime
  oldTime'  <- get oldTime
  let dt = (fromIntegral $ newTime' - oldTime')/200
  react rh (dt, Nothing)
  writeIORef oldTime newTime'
  return ()
```

時間 t を設定し，更新するための関数．この時間 t は main 関数を呼び出してからカウントを開始し，都度新しい t の値を取得することによって他の関数における積分計算において扱うことが可能となる．本研究では関数 t の更新の間隔を 1/200 秒に設定した．

その他，GLUT により描画を行うための関数は以下の通り．

・initGL　描画をするキャンバスの初期設定を行うための関数．ウィンドウの表示や背景色の初期設定，ライティングなどの設定を行う．

・resizeScene　initGL 内で初期設定されるキャンバスのサイズや使用する行列の設定を扱うための関数．この関数でキャンバスのサイズ設定，表示する座標の初期位置の設定（座標の値に合わせてウィンドウ上で見られる領域を設定する），表示する座標のリサイズ，角度，行列を使用する用途の設定を行う．なお，本研究で用いたシミュレーションでは行列の用途は投影となっている．

・clearAndRender　関数 draw から送られたオブジェクトの座標の情報をもとに描画を行うための関数．x 座標と y 座標の値を入力値とし，座標にオブジェクトを描画する．また，描画するオブジェクトの形状や色，影の設定などもこの関数内で行う．

以上の描画用関数に加え，motors 関数にも変更を行う．Haskell ではグローバル変数は実装されておらず，類似した機能を使用する際は Resder モナドなど特定のモナドで代用する必要があるが，本研究ではロボットの制御に関わる部分でなく，またコードの追加部分も少ないため，motors 関数の出力パターンに応じた main 関数，simulate 関数，update 関数を記述し，motors 関数から呼び出す方式とする．

上記の関数をふまえて実装したキャンバスおよびコンソールの画面は以下の通り．

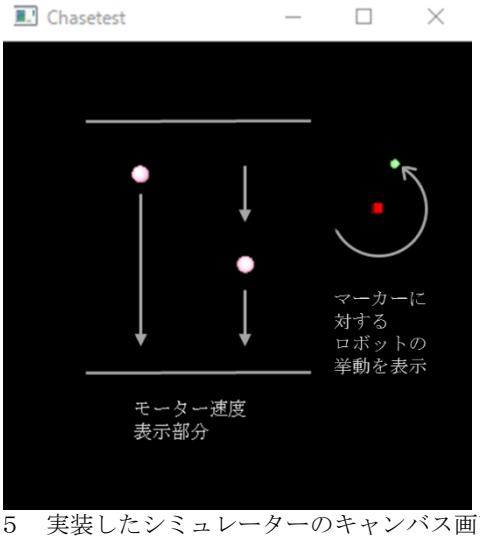

図５　実装したシミュレーターのキャンバス画面

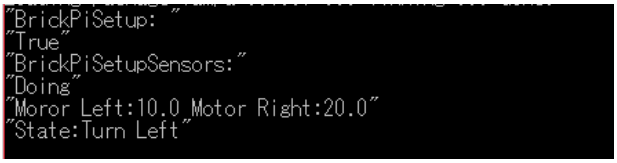

図６　シミュレーターのコンソール画面

キャンバスに表示されたオブジェクトはモーターの速度に応じて-y 方向に等速度直線運動を行い，y=0 の位置で初期位置に戻る．また，キャンバス右上部ではマーカー（赤オブジェクト）に対するロボットの挙動を緑オブジェクトで描画する．なお，キャンバスの描画の関係上モーターの速度（回転数）通りの値を代入してシミュレーションを行うとオブジェクトが消失し正しく描画できなくなるので，今回のプログラムではモーターの速度を 1/10 にして描画を行う．

以上の内容を実装したシミュレーターを実行し，キャンバスの描画とコンソールの表示を併せて入出力が正しく行われているか検証する．

5. 実験結果

Haskell のみを用いたシミュレーションプログラム，および FRP を用いたシミュレーションを実行したときの結果は以下の通り．

対象との距離は超音波センサーの入力値が 20 未満の場合を近い，20 以上 40 未満の場合を中間，40 以上の場合を遠いとする．

表２　Haskell によるシミュレーションの実行結果

ライントレースカー			
障害物	左側床面の色	右側床面の色	ロボットの状態
有	—	—	一時停止
無	黒	黒	速度 50(5.0)/s で直進
無	黒	白	左に旋回
無	白	黒	右に旋回
無	白	白	速度 100(10.0)/s で直進
追従ロボット			
目標	対象との距離（左）	対象との距離（右）	ロボットの状態
無	—	—	追従をせず一時停止
有	近い	近い	一時停止
有	近い	中間	速度 100(10.0)/s で左回転
有	近い	遠い	速度 200(20.0)/s で左回転
有	中間	近い	速度 100(10.0)/s で右回転
有	中間	中間	速度 100(10.0)/s で直進
有	中間	遠い	左に旋回
有	遠い	近い	速度 200(20.0)/s で右回転
有	遠い	中間	右に旋回
有	遠い	遠い	速度 200(20.0)/s で直進

なお，実システムによる検証を行ったところ全てのテストプログラムにおいてセンサーとモーターが起動せず，エラーが起こる現象が発生した．

6. 考察

Haskell によって実装した検証用プログラムは仕様通りセンサーからの入力を想定した値に対し適切なモーターへの出力値を返し，FRP を併せて用いたプログラムでは速度に合わせた描画を行うことができた．また，実システム用のプログラムも入力を直接行うと出力値をコンソールで返すことはできた．一方で，initialize 関数において BrickPi 本体は起動するもののセンサーとモーターは起動せず，エラーが発生した．

従来の研究ではセンサーとモーターが作動したハードを使用した場合でも同様なエラーが発生し，BrickPi が初期設定で使用できる他言語のプログラム(python)で検証したところセンサーとモーター自体は正しく作動したため，今回の検証でロボットが正しく起動しない原因として以下のことが挙げられる．

・従来は正しくセンサーとモーターをセットアップできていた外部インタフェースがセンサーのバージョンアップに対応できていない

・BrickPi の本体および OS，Raspberry Pi の本体およびのアップグレードに外部インタフェースが対応できていない

従来の研究からセンサー，BrickPi の本体および OS，raspberryPi のバージョンアップが行われ実装された Haskell の一部プログラムも修正が行われた中で，外部インタフェースとコンパイル用のプログラムは修正が行われていないため，実システムでのエラーは上記の可能性が高い．

7. むすび

本研究では参照透過性と可読性を持ちながらライントレースカーと追従ロボットのシステム部分をそれぞれ 96 行と 113 行に納めた実装を行い，数値上の入出力の処理は正しく行うことができた．一方で実際のセンサーとモーターに対するセットアップを実装することができず，センサーからの入力や実際にモーターを動かすといった本来実システムで行うべき動作を再現できなかった．今後の課題としてはセンサーやモーター，BrickPi の OS など実システムに直接関わる資材のバージョンアップに対応するための外部インタフェースの修正が挙げられる．また，システム自体は関数を追加することで追従ロボットが目標を見失った場合に別の動作パターンを持たせられる可能性があるが，別途センサーの追加があるため BrickPi が対応できるセンサーの数を越える可能性が高くハードでの課題もある．

外部インタフェースの問題が改善された場合，Haskell の構造的な記述法によってシステムはさらにシンプルな内容で実装できる可能性があるため，今後の研究によってはより複雑な動きをしながらも安定した動作を行いつつ高い可読性を持ったプログラムの実装が期待できる．

文　献

[1] Neil C.C. Brown “Communicating Haskell Processes: Composable Explicit Concurrency Using Monads” Ebook: Communicating Process Architectures 2008, Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. and Broenink, Adam T. Sampson, pp 67-83, IOS Press, Amsterdam, 2008.

[2] “Yampa : Library for programming hybrid systems.”, <https://hackage.haskell.org/package/Yampa>, Online, accessed January 2018

[3] “RaspberryPi － 目 次– bitterharvest’s diary”, <http://bitterharvest.hatenablog.com/entry/2015/12/02/142240> , Online, accessed January 2018

[4] “GLUT - The OpenGL Utility Toolkit”, <https://www.opengl.org/resources/libraries/glut/>, Online, accessed January 2018

[5] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson “Arrows, Robots, and Functional Reactive Programming” *Advanced Functional Programming*, Johan Jeuring, and Simon L. Peyton Jones, pp 159-187, Springer Berlin Heidelberg, Berlin, 2003.

[6] Izzet Pembeci, Henrik Nilsson, and Gregory Hager “Functional Reactive Robotics: An Exercise in Principled Integration of Domain Specific Languages” in *Principles and practice of declarative programming (PPDP ‘02)*, October 2002.

[7] 堀　麻衣, Haskell による複数ロボットの協調動作に関する研究, 情報処理学会第 79 回全国大会, March 2017